```
1: // Computer Program Listing Appendix Under 37 CFR 1.52(e)
2:
3:
4: /*
5:   Appendix includes two sets of code samples, with the first set including:
6: * httprequest_dorequest
7:   HTTPRequest::do_request method.
8: * httprequest_doqueryrequest
9:   HTTPRequest::DoQueryRequest method.
10:   HTTPRequest::do_request does the work of connecting
11:   to the appropriate database, and executing the appropriate
12:   service.
13: * httprequest_parseuri
14:   HTTPRequest::ParseURI method.
15: * httprequest_serviceexists
16:   HTTPRequest::ServiceExists method.
17:   HTTPRequest::ParseURI and HTTPRequest::ServiceExists
18:   do most of the work of parsing a URI to determine which
19:   service maps to that particular URI.
20: * httpconnection_decl
21:   Class declaration for HTTPConnection.
22: * httpprotocol_decl
23:   Class declaration for HTTPProtocol.
24: * httprequest_decl
25:   Class declaration for HTTPRequest.
26: * httppres_decl
27:   Class declaration for HTTPPres.
28: */
29:
30: // httpconnection_decl
31: // Copyright (c) 2004, Sybase, Inc. All Rights Reserved.
32: class HttpConnection {
33:   public:
34:     HttpConnection *  _next;
35:   protected:
36:     HttpSocket *  _socket;
37:     HttpSockOStream *  _stream;
38:     HttpConnectionState  _state;
39:     HttpProtocol *  _protocol;
40:     HttpRequest *  _request;
41:     HttpListener *  _listener;
42:     HttpString    _rmt_addr;
43:     HttpString    _lcl_addr;
44:     HttpString    _rxline;
45:     char *    _dbname;
46:     HttpString    _dbConnected;
47:     char *    _rxbuffer;
48:     char *    _decrypt_buffer;
49:     char    _last;
50:     HttpRxCompletion *  _rxcomplete;
```

```
51:    HttpTxCompletion *  _txcomplete;
52:    uint32    _request_size;
53:    uint32    _pkts_received;
54:    a_fast_tod    _last_read_time;
55:    a_bool    _ignore_receive;
56:    a_bool    _dbname_required;
57:    a_bool    _dbname_provided;
58:    a_web_protocol_type  _type;
59:  public:
60:    HttpConnection( SysSocket sock, char *dbn, a_web_protocol_type
type,
61:        HttpListener *l, char *lcl_addr, char *rmt_addr );
62:    ~HttpConnection();
63:    void    Start( void );
64:    void    Stop( a_bool on_worker = FALSE );
65:    void    DelayedStop( void );
66:    void    RequestFinished( void );
67:    void    CleanUp( void );
68:    void    ProcessSend( int err, int datalen );
69:    a_bool    ProcessLine( char * rxbuffer, int * offset, int datalen
);
70:    void    ProcessRecv( int err, int datalen );
71:    void    ProcessData( char *, int );
72:    void    ProcessHttpsRecv( a_bool force = FALSE );
73:    void    IgnoreReceive();
74:    a_web_protocol_type  GetType( void ) const { return _type; }
75:    char *    GetDbName( void ) const { return _dbname; }
76:    a_bool    DBNameRequired( void ) const { return _dbname_required;
}
77:    a_bool    DBNameProvided( void ) const { return _dbname_provided;
}
78:    char *    GetDbConnected( void ) const { return
_dbConnected.c_str(); }
79:    void    SetDbConnected( char *str ) {
80:  _dbConnected.clear();
81:  _dbConnected.append( str );
82:    }
83:    HttpListener *  GetListener( void ) const { return _listener; }
84:    HttpProtocol *  GetProtocol( void ) const { return _protocol; }
85:    HttpRequest *  GetRequest( void ) const { return _request; }
86:    HttpOrderedList *  GetVariables( void ) const { return
_request->GetVariables(); }
87:    a_bool    ParseRequestString( HttpRequest *request, HttpString
*str );
88:    a_bool    ParseHeaderString( HttpRequest *request, HttpString
*str );
89:    a_bool    ParseURI( HttpRequest * request );
90:    a_bool    ParseVersion( HttpRequest * request );
91:    a_bool    CanDelete( void );
92:    a_bool    CheckForTimeout( void );
```

```
93:     a_bool    SendHttpHeaders( HttpRequest *request, HttpOStream
*stream );
94:     a_bool    SendHttpError( HttpRequest *request, HttpOStream
*stream );
95:     a_bool    SendSQLError( HttpRequest *request, HttpOStream *stream
);
96:     a_bool    IsSecure( void ) const;
97:     void    UpdateReceivedConnProperties( p_Connection ) const;
98:     void    UpdateSentConnProperties( p_Connection ) const;
99:     void    GetRemoteMachineAddr( char *buf, int32 buflen );
100:     uint32    GetIdleTimeout( void ) const;
101:     a_fast_tod    GetLastRequestTime( void ) const { return
_last_read_time; }
102:     HttpString    *GetLocalMachineAddr( void )  { return &_lcl_addr; }
103:     int64    GetBytesWritten( void ) const  { return
_socket->getBytesWritten(); }
104:   protected:
105:     HttpSockOStream *  GetStream() const     { return _stream; }
106:     friend class HttpListener;
107:     friend class HttpProtocol;
108:  private:
109:     void    StartRequest();
110: };
111:
112:
113: // httppres_decl
114: // Copyright (c) 2004, Sybase, Inc. All Rights Reserved.
115: class HttpPres : public StubPres {
116:   protected:
117:     HttpOStream *       _ostream;
118:     UTCollation *  _col;      // db's collation
119:     HttpPresStatus  _status;
120:     HttpOrderedList *  _args;
121:     uint32    _arg_id;
122:     uint32    _arg_len;
123:     a_byte *    _arg_data;
124:     uint32    _descriptor_count;  // number of columns in query
125:     uint32    _row_ctr;    // number of rows written
126:     uint32    _col_ctr;    // number of columns described/written
127:     a_ptrint    _bin_bits;    // which columns are binary data
128:     a_bool    _printed_doc;    // has any doc been written
129:     a_bool    _started_doc;    // has the doc been started
130:     a_bool    _started_set;    // has started the result set
131:     a_bool    _started_row;
132:     a_bool    _wants_null_values;  // output wants to show nulls
133:     a_bool PutData( char ** buf, size_t len, uint32 flags );
134:     a_bool PutData( char * buf, size_t len, uint32 flags )
135:   { return PutData( &buf, len, flags ); }
136:     // Put data Encoded & Charset-converted
137:     inline a_bool PutEC( char * buf )
```

```
138:   { return PutData( buf, _strlen(buf), HF_ENC|HF_CONV ); }
139:     inline a_bool PutEC( char ** buf, size_t len )
140:   { return PutData( buf, len, HF_ENC|HF_CONV ); }
141:     inline a_bool PutEC( char * buf, size_t len )
142:   { return PutData( buf, len, HF_ENC|HF_CONV ); }
143:     inline a_bool PutEC( HttpString & str )
144:   { return PutData( str.str(), str.length(), HF_ENC|HF_CONV ); }
145:     // Put data Charset-converted (no encoding)
146:     inline a_bool PutCC( char ch )
147:   { return PutData( &ch, 1, HF_CONV ); }
148:     inline a_bool PutCC( char * buf )
149:   { return PutData( buf, _strlen(buf), HF_CONV ); }
150:     inline a_bool PutCC( char * buf, size_t len )
151:   { return PutData( buf, len, HF_CONV ); }
152:     inline a_bool PutCC( HttpString & str )
153:   { return PutData( str.str(), str.length(), HF_CONV ); }
154:     // Put ASCII data - not encoded, and not Charset-converted unless
it needs
155:     // to be (i.e. output character set is multi-byte)
156:     inline a_bool PutAsc( char c )
157:   { return PutData( &c, 1, HF_ASC ); }
158:     inline a_bool PutAsc( char * buf )
159:   { return PutData( buf, _strlen(buf), HF_ASC ); }
160:     inline a_bool PutAsc( char * buf, size_t len )
161:   { return PutData( buf, len, HF_ASC ); }
162:     inline a_bool PutAsc( HttpString & str )
163:   { return PutData( str.str(), str.length(), HF_ASC ); }
164:   public:
165:     HttpPres( HttpOStream * ostream, UTCollation * col )
166:   : StubPres()
167:   , _ostream( ostream )
168:   , _col( col )
169:   , _status( PRES_OK )
170:   , _args( NULL )
171:   , _arg_id( 0 )
172:   , _arg_len( 0 )
173:   , _arg_data( NULL )
174:   , _descriptor_count( 0 )
175:   , _row_ctr( 0 )
176:   , _col_ctr( 0 )
177:   , _bin_bits( 0 )
178:   , _printed_doc( FALSE )
179:   , _started_doc( FALSE )
180:   , _started_set( FALSE )
181:   , _started_row( FALSE )
182:   , _wants_null_values( TRUE )
183:     {
184:     }
185:     virtual ~HttpPres();
186:     inline HttpPresStatus GetPresStatus( void )
```

```
187:    {
188:   return _status;
189:    }
190:    inline a_bool PresStatusOk( void )
191:    {
192:   return _status == PRES_OK;
193:    }
194:    void SetArguments( HttpOrderedList * args )
195:    {
196:   // Note: caller responsible for freeing args.
197:   _args = args;
198:    }
199:    a_bool  ReceiveHostVariable( an_sqlpres_value *value, uint32
*index );
200:    a_bool  ReceiveMultiBegin( uint32* total_length );
201:    a_bool  ReceiveMultiPiece( void * buff, uint32 buff_len, uint32*
recv_len );
202:    a_bool  ReceiveMultiEnd( void );
203:    void  ReportSQLError( HttpProtocol * proto );
204:   protected:
205:    a_bool IsBinaryColumn( uint32 c );
206:    inline void SetPresStatus( HttpPresStatus status )
207:    {
208:   if( _status == PRES_OK ) {
209:      _status = status;
210:   }
211:    }
212:    void      MakeErrorString( char * buff, size_t len, a_bool
replace_quotes );
213:   public:
214:    // methods inherited from StubPres
215:    a_bool  SendValueSetDescriptor( uint16    desc_id,
216:          char     *coln_name,
217:          uint16    coln_namelen,
218:          char     *table_name,
219:          char     *db_name,
220:          char     *user_name,
221:          a_byte    asa_domain_id,
222:          uint32    asa_usertype,
223:          uint32    asa_flags,
224:          uint32    asa_maxlen,
225:          uint16    asa_prec,
226:          uint16    asa_scale,
227:          a_describe_flag  describe_flags );
228:    a_bool  ReceiveDescriptor( an_sqlpres_desc * desc );
229:    a_bool  SendValue( a_domain_number domain_id, void * data, uint32
len, uint32 truelen, a_textptr_value * textptr );
230:    a_bool  SendNullValue( a_domain_number domain_id, p_expr expr );
231:    a_bool  SendNoneValue( a_domain_number domain_id );
232:    a_bool  SendMultiBegin( a_domain_number domain_id, uint32
```

```
total_length, uint32 untruncated_length, a_textptr_value * textptr,
uint32 flags );
233:     a_bool  SendMultiPiece( void ** data, uint32 piece_length, uint32
flags );
234:     a_bool  SendMultiEnd( uint32 flags );
235:     a_bool  SendValueSetRow( an_error_mapping *errmap,
an_sqlpres_tran_status tran_status );
236:     a_bool  SendSuccessOrError( p_Connection  conn,
237:             a_bool    send_iocount,
238:             a_bool    send_tran_flags );
239:     a_bool  SendRequestDone( void );
240:     a_bool  SyncPoint( void );
241:     class DBConnConverter *GetConverter( void ) {
242:   return _ostream->getConverter();
243:     }
244:     const class CharsetInfo *GetCharsetInfo( void ) {
245:   return _ostream->getCharsetInfo();
246:     }
247:   protected:
248:     // methods that control output
249:     virtual void AddColumn(
250:       char *    table_name,
251:       char *    coln_name,
252:       uint16    coln_namelen,
253:       uint32    asa_usertype )
254:     {
255:   _unused( table_name );
256:   _unused( coln_name );
257:   _unused( coln_namelen );
258:   _unused( asa_usertype );
259:     };
260:     virtual void BeginDoc( void ) {};    // output doc header
261:     virtual void BeginResultSet( void ) {};  // start of result set
262:     virtual void BeginRow( void ) {};
263:     virtual void BeginColumn( void ) {};
264:     virtual void EndColumn( void ) {};
265:     virtual void EndRow( void ) {};
266:     virtual void EndResultSet( void ) {};
267:     virtual void EndDoc( void ) {};
268:     virtual void SendColumnValue( void * data, uint32 len ) {
_unused(data); _unused(len); };
269:     virtual void SendColumnNull( void ) {};
270:     virtual void SendColumnNone( void ) {};
271:     virtual void SendColumnMultiBegin( void ) {};
272:     virtual void SendColumnMultiPiece( void ** data, uint32 len ) {
_unused(data); _unused(len); };
273:     virtual void SendColumnMultiEnd( void ) {};
274:     virtual void SendSQLError( void * errmsg, size_t len ) = 0;
275:     virtual void NoContentDocBody( void ) {};  // called when doc has
no content
```

```
276:  private:
277:    // these routines guarantee that the virtual versions are called
in correct order
278:    inline void DoBeginDoc( a_bool starting_a_row )
279:    {
280:  if( ! _started_doc ) {
281:      BeginDoc();
282:      if( starting_a_row ) {
283:    BeginResultSet();
284:    _started_set = TRUE;
285:      }
286:      _started_doc = TRUE;
287:      _printed_doc = TRUE;
288:  }
289:    }
290:    inline void DoBeginRow( void )
291:    {
292:  if( ! _started_row ) {
293:      DoBeginDoc( TRUE );    // ensure we've started the doc
294:      BeginRow();
295:      _started_row = TRUE;
296:      _col_ctr     = 0;
297:      _row_ctr ++;
298:  }
299:    }
300:    inline void DoBeginColumn( void )
301:    {
302:  DoBeginRow();      // ensure we have started a row
303:  BeginColumn();
304:    }
305:    inline void DoEndColumn( void )
306:    {
307:  EndColumn();
308:  _col_ctr ++;
309:    }
310:    inline void DoEndRow( void )
311:    {
312:  if( _started_row ) {
313:      EndRow();
314:      _started_row = FALSE;
315:  }
316:    }
317:    inline void DoEndResultSet( void )
318:    {
319:  if( _started_set ) {
320:      DoEndRow();      // ensure we have closed the row
321:      EndResultSet();
322:      _started_set = FALSE;
323:  }
324:    }
```

```
325:     inline void DoEndDoc( void )
326:     {
327:   if( _printed_doc ) {
328:       // we have something on the document
329:       DoEndResultSet();
330:   } else {
331:       // we never generated any doc content
332:       if( ! _started_doc ) {
333:     DoBeginDoc( FALSE );
334:       }
335:       NoContentDocBody();
336:   }
337:   if( _started_doc ) {
338:       EndDoc();
339:       _started_doc = FALSE;
340:   }
341:     }
342: };
343:
344:
345: // httpprotocol_decl
346: // Copyright (c) 2004, Sybase, Inc. All Rights Reserved.
347: class HttpProtocol {
348:   private:
349:     HttpPhase    _phase;    // current phase of the request
350:     a_perf_tod    _tod_connected;  // time of day client connected
351:     a_perf_tod    _tod_queued;  // time of day request was queued
352:     a_perf_tod    _tod_started;  // time of day request was started
353: //  a_perf_tod    _tod_finished;  // time of day request was finished
= time it is logged
354:     HttpString    _method;
355:     HttpString    _uri;
356:     HttpString    _version;
357:     HttpHashTable  _request;  // table of headers in request
358:     HttpHashTable  _response;  // table of headers to send in response
359:     HttpHashTable  _options;  // table of options
360:     HttpString    _lastHeaderKey;  // if value is continued on next
line,
361:                                   // we need to know what to
append it to
362:     HttpStatus    _status;
363:     HttpString    _errorstr;
364:     /*
365:   _body_expected is whether or not we're expecting a body with this
request.
366:   _body_expected_length is the expected length of the body (based on
HTTP headers).
367:   _body.length() is the actual length of the body received so far.
368:     */
369:     HttpString    _body;
```

```
370:    a_bool    _body_expected;
371:    uint32    _body_expected_length;
372:    HttpConnection *  _connection;
373:    HttpLogger *  _logger;
374:    a_bool    _has_been_logged;
375:    void    SetDateHeaders( void );
376:    a_bool    _send_headers;
377:    a_bool    _send_body;
378:    a_bool    _content_type_set;
379:    friend class HttpRequest;
380:    friend class HttpLogger;
381:  public:
382:    HttpProtocol( HttpConnection * connection, HttpLogger * logger );
383:    ~HttpProtocol();
384:    HttpPhase    GetPhase( void )    { return( _phase ); }
385:    a_perf_tod *  GetTodConnected( void )    { return(
&_tod_connected ); }
386:    a_perf_tod *  GetTodQueued( void )    { return( &_tod_queued ); }
387:    a_perf_tod *  GetTodStarted( void )    { return( &_tod_started );
}
388:    void    SetReqQueued( void );
389:    void    SetReqStarted( void );
390:    void    SetReqFinished( void );
391:    void    SetOkToDelete( void );
392:    const HttpString *  GetMethod() const    { return &_method; }
393:    const HttpString *  GetUri() const      { return &_uri; }
394:    const HttpString *  GetVersion() const    { return &_version; }
395:    /*
396:  Request header methods
397:    */
398:    void    SetRequestHeader( const char    *key,
399:            const HttpString  *value ) {
400:  _request.Set( key, value->str(), value->length() );
401:  _lastHeaderKey.clear();
402:  _lastHeaderKey.append( key );
403:    }
404:    void    SetRequestHeader( const char  *key,
405:            const char  *value,
406:            const size_t  len ) {
407:  _request.Set( key, value, len );
408:  _lastHeaderKey.clear();
409:  _lastHeaderKey.append( key );
410:    }
411:    a_bool    AppendRequestHeader( HttpString  &value );
412:    HttpString *  GetRequestHeader( const char * key ) {
413:  return _request.Get( key );
414:    }
415:    HttpString *  GetRequestNextKey( const char * key ) {
416:  return _request.GetNextKey( key );
417:    }
```

```
418:    /*
419:  Response header methods
420:    */
421:    void    SetResponseHeader( const char * key, const HttpString *
value )
422:              { _response.Set( key, value->str(), value->length()
); }
423:    a_bool    SetResponseHeader( const char * key, const char * value
);
424:    HttpString *  GetResponseHeader( const char * key )  { return
_response.Get( key ); }
425:    a_bool    SetHTTPOption( char * optname, char * value );
426:    HttpString *  GetHTTPOption( char * optname )    { return(
_options.Get( optname ) ); }
427:    a_bool    ContentTypeSet( void ) const { return
_content_type_set; }
428:    // _status methods
429:    void    SetHttpStatus( HttpStatus status )  { _status = status; }
430:    void    SetHttpStatus( HttpRequestState state );
431:    HttpStatus    GetHttpStatus( void ) const    { return _status; }
432:    char *    GetHttpStatus( char * buf, size_t len );  // get status
string
433:    void    SetErrorString( const char * str, const size_t len );
434:    const HttpString *  GetErrorString( void ) const    { return
&_errorstr; }
435:    // _body methods
436:    HttpString *  GetBody( void )       { return &_body; }
437:    a_bool    GetBodyExpected( void ) const    { return
_body_expected; }
438:    uint32    GetBodyExpectedLength( void ) const  { return
_body_expected_length; }
439:    a_bool    ParseRequest( const HttpString * request );
440:    a_bool    ParseHeader( const HttpString * header );
441:    a_bool    ParseMethod( void );
442:    a_bool    ParseBodyLength( void );
443:    a_bool    SendHttpHeaders( HttpOStream * stream );
444:    a_bool    SendHttpError( HttpOStream * stream );
445:    void    WriteLogEntry( void );
446:    void    CleanUp( void );
447:    a_bool    ShouldSendBody( void ) const { return _send_body; }
448: #if !PRODUCTION
449:  private:
450:    HttpString    _resbody;
451:  public:
452:    HttpString *  GetResBody( void ) { return &_resbody; }
453: #endif
454: };
455:
456:
457: // httprequest_decl
```

```
458: // Copyright (c) 2004, Sybase, Inc. All Rights Reserved.
459: class HttpRequest : public RQBaseItem
460: {
461:     HttpConnection *  _connection;
462:     HttpProtocol *  _protocol;
463:     HttpOStream *  _stream;
464:     HttpService *  _service;
465:     HttpService *  _dservice;
466:     Database *    _db;
467:     p_Connection  _dbconnection;
468:     p_Worker    _worker;
469:     a_bool    _cancel;
470:     uint32    _uid;
471:     HttpString *        _parms;
472:     HttpRequestState  _state;
473:     HttpString    _username;
474:     HttpString    _password;
475:     HttpString    _database;
476:     HttpString    _service_name;
477:     HttpString    _arguments;
478:     HttpString    _url_path;
479:     HttpOrderedList  _variables;
480:     a_bool    _headers_sent;
481: public:
482:     HttpRequest( HttpConnection * c, HttpProtocol * p, HttpOStream *
s );
483:     ~HttpRequest();
484:     p_Worker    GetWorker() { return _worker; }
485:     HttpOrderedList *  GetVariables() { return &_variables; }
486:     a_bool    Connected() { return _dbconnection != NULL; }
487:     virtual void  do_request();
488:     void    Cancel();
489:     void    CleanUp();
490: private:
491:     a_bool      ServiceExists( HttpString & name );
492:     a_bool      DetermineServiceOptions();
493:     a_bool      ProcessAuthentication();
494:     a_bool      ProcessHttpAuthentication();
495:     a_bool      ProcessBasicAuthentication( const HttpString *
base64_credentials );
496:     a_bool      DatabaseConnect( HttpString &charset );
497:     void      DatabaseDisconnect();
498:     a_bool      ParseURI();
499:     a_bool      ParseArgs( HttpHashTable * argtable, HttpString *
args );
500:     a_bool      ParseMultipartFormData( HttpHashTable * argtable,
HttpString * args, char * boundary );
501:     a_bool      ParseArguments( HttpHashTable * arg_table );
502:     a_bool      ParseBodyArguments( HttpHashTable * arg_table );
503:     a_bool      GetURLPathArguments( HttpHashTable * argtable );
```

```
504:    a_bool     DoDishRequest();
505:    a_bool     DoQueryRequest();
506:    a_bool     DoWSDLRequest();
507:    a_bool     ParseSoapRequest();
508:    void       RedirectToSecure();
509:    void       UpdateReceivedConnProperties( void );
510:    void       UpdateSentConnProperties( void );
511:    void       ReportSQLError( HttpPres * pres );
512:    void       SendHttpHeaders();
513:    void       MakeURI( HttpString * host, HttpService * svc,
HttpString & uri );
514: };
515:
516:
517: // httprequest_doqueryrequest
518: // Copyright (c) 2004, Sybase, Inc. All Rights Reserved.
519: a_bool HttpRequest::DoQueryRequest()
520: /*********************************/
521: {
522:    p_Connection    dbc = _CurrentConnection;
523:    p_statement     stmt = NULL;
524:    HttpPres *      pres = NULL;
525:    p_cursor        crsr = NULL;
526:    p_cursor        real_crsr = NULL;
527:    p_stmt          s = NULL;
528:    p_expr          expr = NULL;
529:    a_bool          variable;
530:    HttpOrderedList stmt_parms;
531: #define CHECK_CANCEL()   if( _cancel ) { goto cleanup; }
532:    if( _service->GetServiceType() == HTTP_SERVICE_SOAP ) {
533:  if( !ParseSoapRequest() ) {
534:      _state = REQUEST_BAD_REQUEST;
535:      return FALSE;
536:  }
537:    } else {
538:  if( !GetURLPathArguments( &_variables ) ) {
539:      _state = REQUEST_BAD_REQUEST;
540:      return FALSE;
541:  }
542:  if( !ParseArguments( &_variables ) ) {
543:      _state = REQUEST_BAD_REQUEST;
544:      return FALSE;
545:  }
546:  if( !ParseBodyArguments( &_variables ) ) {
547:      _state = REQUEST_BAD_REQUEST;
548:      return FALSE;
549:  }
550:    }
551:    CHECK_CANCEL();
552:    /*
```

```
553:   Set up the presentation layer
554:    */
555:    switch( _service->GetServiceType() ) {
556:   case HTTP_SERVICE_XML:
557:      pres = New_HttpPresXML( _stream, dbc->db()->collation );
558:      break;
559:   case HTTP_SERVICE_HTML:
560:      pres = New_HttpPresHTML( _stream,
561:            dbc->db()->collation,
562:            ( _parms == NULL
563:              ? _arguments : *_protocol->GetUri() )
564:           );
565:      break;
566:   case HTTP_SERVICE_RAW:
567:      pres = New_HttpPresRaw( _stream, dbc->db()->collation );
568:      break;
569:   case HTTP_SERVICE_SOAP:
570:      {
571:    HttpString  nspace;
572:    HttpString  opname;
573:    if( _dservice != NULL ) {
574:       MakeURI( (HttpString *)_protocol->_request.Get( "Host" ),
_dservice, nspace );
575:    } else {
576:       MakeURI( (HttpString *)_protocol->_request.Get( "Host" ),
_service, nspace );
577:    }
578:    GetOpName( _dservice, _service, opname  );
579:    pres = New_HttpPresSOAP( _stream, dbc->db()->collation, &nspace,
opname.c_str() );
580:      }
581:      break;
582:   default:
583:      _assertD( FALSE );
584:      break;
585:    }
586:    dbc->pres = pres;
587:    /*
588:   Prepare the statement
589:    */
590:    if( _parms == NULL ) {
591:   // arbitrary query is allowed
592:   expr = an_ExprBuilder::GblBuilder.DB_Expr_str_len(
_arguments.c_str(), (a_row_length) _arguments.length() );
593:    } else {
594:   // service specifies query
595:   expr = an_ExprBuilder::GblBuilder.DB_Expr_str_len( _parms->c_str(),
(a_row_length)_parms->length() );
596:    }
597:    if( expr == NULL ) {
```

```
598:   _state = REQUEST_INTERNAL_ERROR;
599:   goto cleanup;
600:     }
601:     stmt = PrepareExpr( expr, GOAL_STATEMENT, NULL, FALSE );
602:     if( stmt == NULL ) {
603:   DE_Free_expr( expr );
604:   _assertD( SQLErr( dbc ) );
605:   ReportSQLError( pres );
606:   goto cleanup;
607:     }
608:     dbc->SetLastStatement( expr );
609:     DE_Free_expr( expr );
610:     if( stmt->type != STMT_SELECT && stmt->type != STMT_CALL ) {
611:   a_heap_ref  ref;
612:   ref.mem = stmt;
613:   DV_Free_heap( &ref );
614:   _state = REQUEST_BAD_REQUEST;
615:   goto cleanup;
616:     }
617:     if( !SetArgumentNames( &stmt_parms, stmt ) ) {
618:   a_heap_ref  ref;
619:   ref.mem = stmt;
620:   DV_Free_heap( &ref );
621:   _state = REQUEST_SQL_ERROR;
622:   goto cleanup;
623:     }
624:     stmt_parms.CopyValues( &_variables );
625:     CHECK_CANCEL();  // last change to check cancel before doing
actual work
626:     /*
627:   Set up a cursor
628:     */
629:     CreatePreparedStatement( dbc, stmt, &s );
630:     _assertD( !SQLErr( dbc ) );
631:     _assertD( s != NULL );
632:     dbi_describe_statement( s, DESCT_SELECTLIST, DESC_NO_FLAGS, 0,
&variable );
633:     pres->SetArguments( &stmt_parms );
634:     crsr = dbc->add_cursor( "http_cursor" );
635:     crsr->stmt = s;
636:     DBOpenCursor( crsr, -1, CURSOR_READONLY, TRUE );
637:     if( crsr->ref != NULL && !SQLErr( dbc ) ) {
638:   // execute a procedure
639:   dbi_resume_procedure( crsr->ref, TRUE );
640:   if( SQLErr( dbc ) ) {
641:       // proc_cursor has been closed and freed due to error
642:       crsr->proc_cursor = NULL;
643:       crsr->stmt = NULL;
644:   }
645:     }
```

```
646:    real_crsr = dbc->FindRealCursor( crsr, NULL, FALSE );
647:    _state = REQUEST_SUCCESS;
648:    if( real_crsr == NULL ) {
649:   // If statement was procedure call/batch, it might not have had
650:   // a result set.
651:   if( SQLErr( dbc ) ) {
652:       ReportSQLError( pres );
653:   } else {
654:       SendHttpHeaders();
655:   }
656:    } else {
657:   dbc->lock_cursor( real_crsr, TRUE );
658:   dbi_row_descriptor( (p_db_cursor)real_crsr->db_cursor.mem );
659:   /*
660:       Update the status of the HTTP connection and send the headers
661:       out over the wire.
662:   */
663:   _assertD( _state == REQUEST_SUCCESS );
664:   SendHttpHeaders();
665:   if( _protocol->ShouldSendBody() ) {
666:       DoFullFetch( real_crsr );
667:       if( SQLErr( dbc ) ) {
668:     ReportSQLError( pres );
669:       }
670:       switch( pres->GetPresStatus() ) {
671:       case PRES_OSTREAM_ERROR:
672:     _state = REQUEST_FAILURE;
673:     break;
674:       case PRES_NO_XML_USERTYPE:
675:       case PRES_NOT_XML_RESULTSET:
676:     _state = REQUEST_NOT_XML;
677:     break;
678:       }
679:   }
680:   dbc->unlock_cursor( real_crsr );
681:    }
682:    DoCloseCursor( crsr );
683:    dbc->drop_statement( s );
684:    if( _protocol->ShouldSendBody() ) {
685:   dbc->pres->SendSuccessOrError( dbc, TRUE, TRUE );
686:    }
687: #undef CHECK_CANCEL
688:  cleanup:
689:    if( pres != NULL ) {
690:   dbc->pres = NULL;
691:   delete pres;
692:    }
693:    if( _cancel ) {
694:   _state = REQUEST_CANCELED;
695:    }
```

```
696:     return _state == REQUEST_SUCCESS;
697: }
698:
699:
700: // httprequest_dorequest
701: // Copyright (c) 2004, Sybase, Inc. All Rights Reserved.
702: void HttpRequest::do_request()
703: /*****************************/
704: {
705:     UserDef  *user = NULL;
706:     HttpString  charset( 100 );
707:     _worker = _CurrentWorker;
708:     _protocol->SetReqStarted();
709: #define CHECK_CANCEL()   if( _cancel ) { goto finish; }
710:     CHECK_CANCEL();
711:     if( !ParseURI() ) {
712:   _state = REQUEST_BAD_REQUEST;
713:   goto finish;
714:     }
715:     if( !DatabaseConnect( charset ) ) {
716:   goto finish;
717:     }
718:     if( !DetermineServiceOptions() ) {
719:   goto finish;
720:     }
721:     /*
722:   Depending on the URI, the DISH service either acts like a WSDL
723:   service or a SOAP service.
724:     */
725:     if( _service->GetServiceType() == HTTP_SERVICE_DISH ) {
726:   if( _arguments.eq( "wsdl" ) ) {
727:       /*
728:     "Fall through".  The DISH service is generating WSDL, and will be
729:     treated as a WSDL service below.
730:       */
731:   } else {
732:       /*
733:     Set _service_name to the service specified in SOAPAction.
734:     SOAPAction URLs have the format "http://hostname/dbname/path"
735:     (quotes included).
736:       */
737:       HttpString * action = _protocol->GetRequestHeader( "SOAPAction"
);
738:       if( action != NULL ) {
739:     HttpStrIStream  stream( action );
740:     int    slash = 0;
741:     unsigned char  c;
742:     while( TRUE ) {
743:         if( !stream.get( c ) ) goto finish;
744:         if( c == '/' ) {
```

```
745:       if( ++slash >= 4 ) break;
746:         }
747:     }
748:     _service_name.clear();
749:     while( TRUE ) {
750:         if( !stream.get( c ) ) goto finish;
751:         if( c == "" ) break;
752:         _service_name.append( c );
753:     }
754:     _dservice = _service;
755:     if( !DetermineServiceOptions() ) {
756:         goto finish;
757:     }
758:     /*
759:         DISH won't act as a proxy for anything other than SOAP
services.
760:     */
761:     if( _service->GetServiceType() != HTTP_SERVICE_SOAP ) {
762:         goto finish;
763:     }
764:       }
765:   }
766:     }
767:     /*
768:   Secure connection
769:     */
770:     if( _service->GetSecureRequired() && !_connection->IsSecure() ) {
771:   RedirectToSecure();
772:   goto finish;
773:     }
774:     /*
775:   Authentication
776:     */
777:     if( _service->GetAuthRequired() ) {
778:   if( !ProcessAuthentication() ) {
779:       goto finish;
780:   }
781:     }
782:     user = FindUserByID( _uid );
783:     if( user == NULL ) {
784:   _state = REQUEST_INVALID_USER;
785:   goto finish;
786:     }
787:     _dbconnection->SetUser( user->GetSAUserName(), FALSE, TRUE );
788:     _dbconnection->set_user( user );
789:     if( AuditingOn( _db ) ) {
790:   char      address[80];
791:   a_web_protocol_type  proto = WEB_HTTP;
792:   _connection->GetRemoteMachineAddr( address,
793:             (int32)sizeof( address ) );
```

```
794:   if( _connection->IsSecure() ) {
795:       proto = WEB_HTTPS;
796:   }
797:   AuditHttpConnection( user->GetSAUserName(), address, GetProtoStr(
proto ),
798:           TRUE, _db );
799:     }
800:     user->Release();
801:     CHECK_CANCEL();
802:     if( !CallLoginEnvironment( _dbconnection ) ) {
803:   DB_Exec_connect_failed_event_handler( _db,
804:       (char *)_dbconnection->_ew_static_user.str(), NULL );
805:   _state = REQUEST_INVALID_AUTHENTICATION;
806:   goto finish;
807:     }
808:     if( Debug ) {
809:   DB_Message( IDS_ENG_USER_CONNECTED_TO_DATABASE_FROM_HTTP,
810:         _dbconnection->handle(),
811:         _dbconnection->get_user()->name,
812:         _db->_ro_alias,
813:         GetProtoStr( _connection->GetType() ) );
814:   DBConnConverter *conv = _stream->getConverter();
815:   if( conv == NULL ) {
816:       DB_Message(
IDS_ENG_CHARSET_TRANSLATION_ENABLED_NOT_NEEDED_WITH_CONNID,
817:             _dbconnection->handle(),
818:       _db->cs_info->sybase_label );
819:   } else {
820:       const CharsetInfo *cli_cs_info =
821:     UTLocale::GetCharsetInfoFromAsaCID( conv->outbound().GetDestCid()
);
822:       DB_Message(
IDS_ENG_CHARSET_TRANSLATION_ENABLED_NEEDED_WITH_CONNID,
823:       _dbconnection->handle(),
824:       _db->cs_info->sybase_label,
825:       cli_cs_info->sybase_label );
826:   }
827:     }
828:     DB_Exec_system_event_handler( _dbconnection->db(), _dbconnection,
EVT_Connect );
829:     ProcDebug::ConnectionStarted( _dbconnection );
830:     /*
831:   Define special request header fields so user can access them.
832:   BUGBUG: What do we set URI to when processing a SOAP request through
833:     a DISH service?  For now, we're setting it to the URI of the
834:     DISH service, not the proxied SOAP service.
835:     */
836:     _protocol->SetRequestHeader( "@HttpMethod",
_protocol->GetMethod() );
837:     _protocol->SetRequestHeader( "@HttpURI", _protocol->GetUri() );
```

```
838:     _protocol->SetRequestHeader( "@HttpVersion",
_protocol->GetVersion() );
839:     UpdateReceivedConnProperties();
840:     CHECK_CANCEL();
841:     switch( _service->GetServiceType() ) {
842:   case HTTP_SERVICE_XML:
843:   case HTTP_SERVICE_HTML:
844:   case HTTP_SERVICE_RAW:
845:   case HTTP_SERVICE_SOAP:
846:       DoQueryRequest();
847:       break;
848:   case HTTP_SERVICE_DISH:
849:   case HTTP_SERVICE_WSDL:
850:       DoWSDLRequest();
851:       break;
852:   default:
853:       _assertD( FALSE );
854:       _state = REQUEST_INVALID_SERVICE;
855:       break;
856:     }
857: #undef CHECK_CANCEL
858: finish:
859:     DatabaseDisconnect();
860:     if( _cancel ) {
861:   _state = REQUEST_CANCELED;
862:     }
863:     if( _state != REQUEST_SUCCESS ) {
864:   _protocol->SetHttpStatus( _state );
865:   if( _state == REQUEST_SQL_ERROR ) {
866:   } else {
867:       _protocol->SendHttpError( _stream );
868:   }
869:     }
870:     _stream->flush();
871:     UpdateSentConnProperties();
872:     _worker = NULL;
873:     _connection->RequestFinished();
874: }
875:
876:
877: // httprequest_parseuri
878: // Copyright (c) 2004, Sybase, Inc. All Rights Reserved.
879: a_bool HttpRequest::ParseURI()
880: /*****************************/
881: {
882:     HttpStrIStream  stream( _protocol->GetUri() );
883:     unsigned char   c;
884:     a_bool      got_service = FALSE;
885:     _assertD( _database.length() == 0 );
886:     _assertD( _service_name.length() == 0 );
```

```
887:     _assertD( _arguments.length() == 0 );
888:     if( !stream.get( c ) || c != '/' ) {
889:   return FALSE;
890:     }
891:     if( _connection->DBNameProvided() ) {
892:   _database.append( _connection->GetDbName() );
893:     } else {
894:   while( TRUE ) {
895:       if( !stream.get( c ) ) return TRUE;
896:       if( c == '/' ) break;
897:       if( c == '?' ) {
898:     got_service = TRUE;
899:     break;
900:       }
901:       _database.append( c );
902:   }
903:     }
904:     if( !got_service ) {
905:   while( TRUE ) {
906:       if( !stream.get( c ) ) return TRUE;
907:       if( c == '?' ) break;
908:       _service_name.append( c );
909:   }
910:     }
911:     while( stream.get( c ) ) {
912:   _arguments.append( c );
913:     }
914:     return TRUE;
915: }
916:
917:
918: // httprequest_serviceexists
919: // Copyright (c) 2004, Sybase, Inc. All Rights Reserved.
920: a_bool HttpRequest::ServiceExists( HttpString & name )
921: /*****************************************************/
922: {
923:     a_bool        result = FALSE;
924:     HttpService *      svc;
925:     a_statement *      stmt;
926:     p_Database        db = _CurrentDB;
927:     svc = HttpService::Find( db, name.c_str() );
928:     if( svc == NULL ) {
929:   // The service by the full name does not exist...
930:   // need to split it up into <name>/<url> pieces
931:   char *        str   = name.c_str();
932:   size_t        len   = name.length();
933:   if( len == 0 ) {
934:       return FALSE;
935:   }
936:   size_t        split = len - 1;
```

```
937:   for( ; split > 0; split-- ) {   // Note: first char cannot be '/'
938:       if( str[split] == '/' ) {
939:     // split at this point
940:     HttpString tname( str, split );
941:     svc = HttpService::Find( db, tname.c_str() );
942:     if( svc != NULL ) {
943:         if( svc->GetUrlPathType() != URL_PATH_OFF ) {
944:       // got [name=0..split-1]/[url=split+1..len-1]
945:       len = len - split - 1;
946:       if( len == 0 || svc->GetUrlPathType() == URL_PATH_OFF ) {
947:           _url_path.set_empty();
948:       } else {
949:           _url_path.append( str+(split+1), len );
950:       }
951:       name.resize( split );
952:       break;
953:         } else {
954:       svc->Release();
955:       svc = NULL;
956:         }
957:     }
958:       }
959:   }
960:   if( split == 0 ) {
961:       return FALSE;
962:   }
963:     }
964:     if( svc != NULL ) {
965:   _uid = svc->GetUid();
966:   stmt = svc->LockStmt();
967:   if( stmt != NULL ) {
968:       p_expr  stmtstr;
969:       uint32  len;
970:       stmtstr = Prep_to_str( NULL, stmt );
971:       stmtstr = an_ExprBuilder::GblBuilder.DB_Find_expr( stmtstr,
FALSE );
972:       len = (uint32) stmtstr->v.str->length();
973:       _parms = new HttpString( len + 1 );
974:       {
975:     // Copy string to _parms
976:     DbStrlStream  s( *stmtstr->v.str, _CurrentConnection );
977:     s.get( (a_byte *) _parms->str(), (uint32)
stmtstr->v.str->length() );
978:     _parms->resize( len );
979:       }
980:       DE_Free_expr( stmtstr );
981:   }
982:   svc->UnlockStmt();
983:   result = TRUE;
984:     }
```

```
985:     _service = svc;
986:     return result;
987: }
988:
989: // httppresxml
990: // Copyright (c) 2004, Sybase, Inc. All Rights Reserved.
991: // *****************************************************************
992: // Copyright 2002-2003 iAnywhere Solutions, Inc.  All rights reserved.
993: // *****************************************************************
994: #include "httppres.h"
995: #include "httpstring.h"
996: #include "dbusrtyp.h"
997: #include "httputil.h"
998:   // #include "dblangstring.hpp"
999: #include "cachecarver.hpp"
1000: #include "dbvector.h"
1001: typedef struct _a_col_name {
1002:     struct _a_col_name * next;
1003:     uint16      len;
1004:     char        name[2];      // a variable length
1005: } a_col_name, *p_col_name;
1006: //*************************************************************************
**
1007: class HttpPresXML : public HttpPres {
1008: protected:
1009:     UTCollation *    _col;      // db's collation
1010:     a_bool      _has_xml;      // db has XML typeid /
1011:     uint16      _xml_typeid;
1012:     a_bool      _saw_xml;      // we saw some XML columns
1013:     a_bool      _do_xml_formatting;    // we need to do the formatting
1014:     CacheCarver *   _carver;
1015:     p_col_name      _first_col;
1016:     p_col_name      _current;    // "cur" column i.e. column[_cur_idx]
1017:     uint16          _cur_idx;
1018: public:
1019:     HttpPresXML( HttpOStream * ostream, UTCollation * col );
1020:     virtual ~HttpPresXML();
1021: protected:
1022:     virtual void AddColumn(
1023:       char *    table_name,
1024:       char *    coln_name,
1025:       uint16    coln_namelen,
1026:       uint32    asa_usertype );
1027:     virtual void BeginDoc( void );
1028:     virtual void BeginResultSet( void );  // start of result set
(table)
1029:     virtual void BeginRow( void );
1030:     virtual void BeginColumn( void );
1031:     virtual void EndColumn( void );
1032:     virtual void EndRow( void );
```

```
1033:     virtual void EndResultSet( void );    // end result set (table)
1034:     virtual void EndDoc( void );
1035:     virtual void SendColumnValue( void * data, uint32 len );
1036:     virtual void SendColumnMultiPiece( void * data, uint32 len );
1037:     virtual void SendSQLError( void * errmsg, size_t len );
1038:     virtual void NoContentDocBody( void );  // called when doc has no
content
1039: };
1040: //************************************************************************
**
1041: HttpPres * New_HttpPresXML( HttpOStream * ostream, UTCollation * col )
1042: /************************************************************************/
1043: {
1044:     return new HttpPresXML( ostream, col );
1045: }
1046: HttpPresXML::HttpPresXML( HttpOStream * ostream, UTCollation * col )
1047:     : HttpPres( ostream, col )
1048:     , _has_xml( FALSE )
1049:     , _xml_typeid( 0 )
1050:     , _saw_xml( FALSE )
1051:     , _do_xml_formatting( FALSE )
1052:     , _carver( NULL )
1053:     , _first_col( NULL )
1054:     , _current( NULL )
1055:     , _cur_idx( 0 )
1056: /*******************************/
1057: {
1058:     a_user_type * utype = FindUserType( "xml" );
1059:     if( utype != NULL ) {
1060:   _has_xml    = TRUE;
1061:   _xml_typeid = utype->type_id;
1062:     } else {
1063:   //  SetPresStatus( PRES_NO_XML_USERTYPE );
1064:   // we will format the XML output in this code
1065:   _do_xml_formatting = TRUE;
1066:     }
1067:     _wants_null_values = FALSE;   // XML does not show null values
1068: }
1069: HttpPresXML::~HttpPresXML()
1070: /*************************/
1071: {
1072:     if( _carver != NULL ) {
1073:   delete _carver;
1074:     }
1075: }
1076: //************************************************************************
**
1077: // Document generation routines
1078: //************************************************************************
**
```

```
1079: void HttpPresXML::AddColumn(
1080:       char *    table_name,
1081:       char *    coln_name,
1082:       uint16    coln_namelen,
1083:       uint32    asa_usertype )
1084: /*****************************************************/
1085: {
1086:     _unused( table_name );
1087:     _unused( coln_name );
1088:     _unused( coln_namelen );
1089:     if( _carver == NULL ) {
1090:   _carver    = new CacheCarver( NULL );
1091:     }
1092:     // check that the column name is valid XML name
1093:     // if the user did not format the output as XML
1094:     // then we are only going to allow "nice" ascii names
1095:     int    i;
1096:     int    k;
1097:     p_col_name  p;
1098:     char  cname[256];
1099:     for( i=0; i<coln_namelen; i++ ) {
1100:   if( coln_name[i] >= 'a' && coln_name[i] <= 'z' ) continue;
1101:   if( coln_name[i] >= 'A' && coln_name[i] <= 'Z' ) continue;
1102:   if( coln_name[i] == '_' || coln_name[i] == ':' ) continue;
1103:   if( i == 0 ) break;
1104:   if( coln_name[i] >= '0' && coln_name[i] <= '9' ) continue;
1105:   if( coln_name[i] == '-' || coln_name[i] == '.' ) continue;
1106:   break;
1107:     }
1108:     if( i < coln_namelen || coln_namelen == 0 ) {
1109:   // not a valid name - make one up
1110:   sprintf( cname, "_%d", _descriptor_count );
1111:   coln_name    = cname;
1112:   coln_namelen = _strlen( cname );
1113:     }
1114:     // make sure that the column name is unique
1115:     for( k=0,i=0,p=_first_col; p!=NULL; ) {
1116:   if( p->len == coln_namelen ) {
1117:       if( _strnieq( p->name, coln_name, coln_namelen ) ) {
1118:     // duplicate name - use a unique name
1119:     sprintf( cname, ((i==0)?"_%d_%d":"_%d_%d_%d"), _descriptor_count,
k+1, i++ );
1120:     coln_name    = cname;
1121:     coln_namelen = _strlen( cname );
1122:     // start over again to make sure our new name is now unique
1123:     k = 0;
1124:     p = _first_col;
1125:       }
1126:   }
1127:   k++;
```

```
1128:   p = p->next;
1129:     }
1130:     // when we get here, the name is unique and so add it after
_current
1131:     p = (p_col_name)_carver->alloc( AL_MEMORY,
sizeof(a_col_name)+coln_namelen );
1132:     memcpy( p->name, coln_name, coln_namelen );
1133:     p->next = NULL;
1134:     p->len  = coln_namelen;
1135:     if( _first_col == NULL ) {
1136:   _first_col = p;
1137:   _cur_idx   = 0;
1138:     } else {
1139:   _current->next = p;
1140:   _cur_idx ++;
1141:     }
1142:     _current = p;
1143:     if( _has_xml && asa_usertype == _xml_typeid ) {
1144:   _saw_xml = TRUE;
1145:     }
1146:     if( _descriptor_count == 1 && asa_usertype == _xml_typeid ) {
1147:   // we only expect to get one XML column ...
1148:   // if we don't, then we will format the columns into XML ourselves
1149:     } else {
1150:   //  SetPresStatus( PRES_NOT_XML_RESULTSET );
1151:   // force formatting of the output in this code
1152:   _do_xml_formatting = TRUE;
1153:     }
1154: }
1155: void HttpPresXML::BeginDoc( void )
1156: /*******************************/
1157: {
1158:     PutAsc( "<?xml version=\"1.0\" ?>\n" );
1159: }
1160: void HttpPresXML::BeginResultSet( void )
1161: /*************************************/
1162: {
1163:     PutAsc( "<root>\n" );
1164: }
1165: void HttpPresXML::BeginRow( void )
1166: /*********************************/
1167: {
1168:     if( _do_xml_formatting ) {
1169:   PutAsc( "<row" );
1170:   _current = _first_col;
1171:   _cur_idx = 0;
1172:     }
1173: }
1174: void HttpPresXML::BeginColumn( void )
1175: /***********************************/
```

```
1176: {
1177:     if( _do_xml_formatting ) {
1178:   for(; _cur_idx < _col_ctr; _cur_idx++ ) {
1179:       _assertD( _current != NULL );
1180:       _current = _current->next;
1181:   }
1182:   _assertD( _current != NULL );
1183:   PutAsc( ' ' );
1184:   PutData( _current->name, _current->len, HF_COLNAME );
1185:   PutAsc( "=\"" );
1186:     }
1187: }
1188: void HttpPresXML::EndColumn( void )
1189: /*********************************/
1190: {
1191:     if( _do_xml_formatting ) {
1192:   PutAsc( "" );
1193:     }
1194: }
1195: void HttpPresXML::EndRow( void )
1196: /*******************************/
1197: {
1198:     if( _do_xml_formatting ) {
1199:   PutAsc( "/>\n" );
1200:     }
1201: }
1202: void HttpPresXML::EndResultSet( void )
1203: /************************************/
1204: {
1205:     PutAsc( "</root>\n" );
1206: }
1207: void HttpPresXML::EndDoc( void )
1208: /******************************/
1209: {
1210: }
1211: void HttpPresXML::SendColumnValue( void * data, uint32 len )
1212: /************************************************************/
1213: {
1214:     if( _do_xml_formatting ) {
1215:   if( len > 0 ) {
1216:       PutData( (char *)data, (size_t)len, HF_ENC );  // db data is
cs-conv up in engine
1217:   }
1218:     } else {
1219:   // no encoding or conversion is required because the engine already
did it
1220:   PutData( (char *)data, (size_t)len, HF_NONE );    // data is
already XML
1221:     }
1222: }
```

```
1223: void HttpPresXML::SendColumnMultiPiece( void * data, uint32 len )
1224: /*************************************************************/
1225: {
1226:     if( _do_xml_formatting ) {
1227:   if( len > 0 ) {
1228:       PutData( (char *)data, (size_t)len, HF_ENC );  // db data is
cs-conv up in engine
1229:   }
1230:     } else {
1231:   // no encoding or conversion is required because the engine already
did it
1232:   PutData( (char *)data, (size_t)len, HF_NONE );    // data is
already XML
1233:     }
1234: }
1235: void HttpPresXML::SendSQLError( void * errmsg, size_t len )
1236: /*************************************************************/
1237: {
1238:     PutAsc( "<SQLerror message=\"" );
1239:     PutData( (char *)errmsg, len, HF_ENC|HF_CONV );
1240:     PutAsc( "\"/> " );
1241: }
1242: void HttpPresXML::NoContentDocBody( void )
1243: /*****************************************/
1244: {
1245:     // in this case, we want to just dump the headers without any rows
1246:     BeginResultSet();
1247:     EndResultSet();
1248: }
1249:
1250:
1251: // sa_set_http_header
1252: // Copyright (c) 2004, Sybase, Inc. All Rights Reserved.
1253: static a_ptrint
1254: sa_set_http_header( a_queue *parm_q )
1255: /***********************************/
1256: {
1257:     char *  fldname;
1258:     char *  val;
1259:     GetParms( parm_q, &fldname, &val );
1260: #if defined( HTTP_SUPPORT )
1261:     if( !_CurrentConnection->SetHTTPHeaderField( fldname, val ) ) {
1262:   dbi_sql_errors( SQLSTATE_INVALID_HTTP_HEADER_SETTING, fldname );
1263:   return( -1 );
1264:     }
1265: #endif
1266:     return( 0 );
1267: }
1268:
1269:
```

```
1270: // sa_set_http_option
1271: // Copyright (c) 2004, Sybase, Inc. All Rights Reserved.
1272: static a_ptrint
1273: sa_set_http_option( a_queue *parm_q )
1274: /***********************************/
1275: {
1276:     char *  optname;
1277:     char *  val;
1278:     GetParms( parm_q, &optname, &val );
1279: #if defined( HTTP_SUPPORT )
1280:     if( !_CurrentConnection->SetHTTPOption( optname, val ) ) {
1281:   dbi_sql_errors( SQLSTATE_INVALID_HTTP_OPTION_SETTING, optname );
1282:   return( -1 );
1283:     }
1284: #endif
1285:     return( 0 );
1286: }
1287:
1288:
1289: // sethttpheaderfield
1290: // Copyright (c) 2004, Sybase, Inc. All Rights Reserved.
1291: a_bool Connection::SetHTTPHeaderField( char * fldname, char * val )
1292: /*****************************************************************/
1293: {
1294:     a_bool  result = FALSE;
1295:     if( http_conn != NULL && fldname != NULL && *fldname != '\0' ) {
1296:   result = http_conn->GetProtocol()->SetResponseHeader( fldname, val
== NULL ? "" : val );
1297:     }
1298:     return( result );
1299: }
1300:
1301:
1302: // sethttpoption
1303: // Copyright (c) 2004, Sybase, Inc. All Rights Reserved.
1304: a_bool Connection::SetHTTPOption( char * optname, char * val )
1305: /************************************************************/
1306: {
1307:     a_bool  result = FALSE;
1308:     if( http_conn != NULL && optname != NULL && *optname != '\0' ) {
1309:   result = http_conn->GetProtocol()->SetHTTPOption( optname,
1310:     (char *)( val == NULL ? "" : val ) );
1311:     }
1312:     return( result );
1313: }
1314: a_bool HttpProtocol::SetHTTPOption( char * optname, char * value )
1315: /*****************************************************************/
1316: {
1317:     // Validate option names [and values].
1318:     if( _strieq( optname, "CharsetConversion" ) ) {
```

```
1319:   if( _strieq( value, "ON" ) ) {
1320:     _connection->GetStream()->set_translation_wanted( TRUE );
1321:   } else if( _strieq( value, "OFF" ) ) {
1322:     _connection->GetStream()->set_translation_wanted( FALSE );
1323:   } else {
1324:     return( FALSE );
1325:   }
1326:    } else {
1327:   return( FALSE );
1328:    }
1329:    _options.Set( optname, value );
1330:    return( TRUE );
1331: }
1332:
1333:
1334: // setresponseheader
1335: // Copyright (c) 2004, Sybase, Inc. All Rights Reserved.
1336: a_bool HttpProtocol::SetResponseHeader( const char * key, const char
* value )
1337: /*************************************************************************
*******/
1338: {
1339:    _assertD( key != NULL );
1340:    if( *key == '@' ) {
1341:   // special values
1342:   if( _strieq( key, "@HttpStatus" ) ) {
1343:     // value had better be the numeric status code
1344:     int v = atoi( value );
1345:     int i = StatusLineIndex( v );
1346:     if( i == LEVEL_600 ) {
1347:    // invalid value
1348:    return FALSE;
1349:     }
1350:     SetHttpStatus( (HttpStatus)v );
1351:     return TRUE;
1352:   }
1353:    }
1354:    // check that the key and value consists of valid HTTP characters
1355:    const char * s;
1356: #define IS_HTTP_TOKEN_CHAR( c ) ( (c)>' ' && (c)<='~' &&
!IsHttpSeparator( c ) )
1357: #define VALID_HTTP_KEY_CHAR( c ) IS_HTTP_TOKEN_CHAR( c )
1358: #define VALID_HTTP_VAL_CHAR( c ) ( ((c)>=' ' && (c)<='~') ||
(c)=='\t' )
1359:    // TBD: we need to handle LWS (continuation lines) in the field
values
1360:    for( s=key; *s != '\0'; s++ ) {
1361:   if( !VALID_HTTP_KEY_CHAR( *s ) ) {
1362:     return FALSE;
1363:   }
```

```
1364:     }
1365:     for( s=value; *s != '\0'; s++ ) {
1366:   if( !VALID_HTTP_VAL_CHAR( *s ) ) {
1367:       return FALSE;
1368:   }
1369:     }
1370:     _response.Set( key, value );
1371:     return TRUE;
1372: }
1373:
1374:
```